User-Defined Data Distributions in High-Level Programming Languages

Roxana E Diaconescu
Center for Advanced Computing Research
California Institute of Technology
Pasadena, CA 91125
Email: roxana@caltech.edu

Hans P. Zima Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
Email: zima@jpl.nasa.gov

Abstract

One of the characteristic features of today's high performance computing systems is a physically distributed memory. Efficient management of locality is essential for meeting key performance requirements for these architectures. The standard technique for dealing with this issue has involved the extension of traditional sequential programming languages with explicit message-passing, in the context of a processor-centric view of parallel computation. This has resulted in complex and error-prone assembly-style codes in which algorithms and communication are inextricably interwoven.

This paper presents a high-level approach to the design and implementation of data distributions. Our work is motivated by the need to improve the current parallel programming methodology by introducing a paradigm supporting the development of efficient and reusable parallel code. This approach is currently being implemented in the context of a new programming language called Chapel, which is designed in the HPCS project Cascade.

1 Introduction

A key feature of today's high performance computing systems is a physically distributed memory, which is common to all architectures on the Top500 list [16]. Efficient management of locality is essential for these machines. The standard technique for dealing with this issue has involved the extension of traditional sequential programming languages such as Fortran, C, and C++ with explicit message-passing, in the context of a processor-centric view of parallel computation. This has resulted in complex and error-prone programs in which algorithms and communication are inextricably interwoven.

The High Performance Fortran (HPF) family of languages proposed a higher-level programming paradigm based on an abstract specification of data distribution by the programmer while delegating the generation of explicit message passing code to the compiler/runtime system. This approach presented a major step towards more expressive parallel languages, but has not, for a variety of reasons, been broadly accepted by the user community. However, it has deeply influenced later research, including the work presented in this paper. We describe the design of a powerful facility for *defining* new data distributions in the context of the Chapel programming language [3, 8]. User-defined distributions increase the power of the underlying language similar to the way function definitions raise the operational level of a programming language: new data distributions can be generated as first-class objects in a language-provided framework, placed in a library, passed to functions, and reused in array declarations. In the simplest case, the specification of a new distribution can consist of just a few lines of code to define a mapping from global indices to memory; in contrast, a sophisticated user (or distribution writer) can control the internal representation and layout of data to an almost arbitrary degree, allowing even the expression of auxiliary structures typically used for distributed sparse matrix data.

Our goal is to provide maximum flexibility to the programmer when distributing data collections across *locales* (units of uniform memory access), while enabling compiler transformations and optimizations that deal with low-level details of distribution management such as explicit distinction between local and remote accesses and generation of communication and synchronization. The programmer retains control of the higher-level aspects of data distribution: the spe-

2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)
0-7695-2644-6/06 $20.00 © 2006 IEEE

cific challenge for the design of the relevant language features is to come up with a set of primitives and interfaces which establish a productive communication between programmer intentions and compiler transformations.

The rest of the paper is organized as follows. After summarizing related research efforts and their relationship to our work in Section 2 we establish the conceptual foundation for the discussion of distributions in Section 3. The following Section 4 shows how distributions can be defined and used in Chapel, illustrating the our framework for specifying new distributions with a set of simple examples. Finally, Section 5 states the main conclusion of our paper and summarizes future directions of research.

2 Related Work

Our work builds on research performed by many groups over the past decades. IVTRAN [22], developed for the SIMD architecture ILLIAC IV, was an early language providing high-level control of data distributions. With the advance of distributed-memory systems in the 1980s, a new class of *data-parallel languages* was explored. In such languages the large data structures in an application are laid out across the memories of a distributed-memory parallel machine. The subcomponents of these distributed data structures can then be operated upon in parallel on all processors. Kali [21] was the first language to introduce distribution declarations in the context of distributed-memory architectures, together with a simple mechanism for user-defined distributions. Fortran D [14], Vienna Fortran [5], and Connection Machine Fortran [1], the major predecessors of the High Performance Fortran [15] effort, all offered facilities for combining multi-dimensional array declarations with the specification of a data distribution or alignment. Several other academic as well as commercial projects also contributed to the understanding necessary for the development of data parallel languages and the required compilation technology.

These languages largely relied on a set of built-in mechanisms, such as regular *block* and *block-cyclic* distributions, as well as limited features for irregular distributions, such as *general block* and *indirect*. The Vienna Fortran language specification [18] introduced a capability for user-defined mappings from Fortran arrays to a set of abstract processors, and for user-defined alignments between arrays. Distribution classes more general than the standard distributions mentioned above include the Kelp library [13] and the generalized multipartitioning scheme implemented in Rice University's dHPF compiler [9]. More recently, High Performance Java [20] and a number of parallel Matlab versions [7] have extended their base languages with high-level distribution specifications.

Object-oriented language extensions and systems such as ICC++ [6] and pC++ [2] wrap distributions and data into classes and collections with overridable behavior to account for reuse and gain flexibility. This represents significant progress with respect to productivity, but no advance regarding the specification and efficient utilization of distributions. Distributions are either restricted to a set of built-in types, or are specified via restrictive mechanisms (e.g., in ICC++ a map file which is a sequence of integer indices along with virtual processor numbers needs to be supplied manually). Charm++ [19] lets the runtime system decide how to map objects to processors such as to ensure load-balance.

Recent language developments include the emerging class of partitioned global array (PGAS) languages, such as CoArray Fortran [10], Unified Parallel C [17], and Titanium [24]. These languages provide standard distributions but still require the user to explicitly control communication in the context of a processor-centric programming model. Thus they represent an advance with regard to the MPI-based programming paradigm, but do not target a broader productivity impact along the lines of the Chapel approach.

Closer to the goals represented by Chapel are two languages developed along with Chapel in DARPA's High Productivity Computing Systems (HPCS) program: X10, designed in the PERCS project led by IBM [12, 23], and Fortress [11], developed at SUN Microsystems. Both languages use the HPF-inspired approach of providing explicit data distribution declarations.

A key difference between the language work reported above and our research is that we do not study new partitioning strategies for inclusion into a set of distributions offered by the language. There is no concept of built-in distribution in Chapel: we provide a general method for allowing novel distributions to be expressed without modifying the compiler. A predefined generic distribution type can be customized by the programmer to express specific needs of the application, algorithms, or data access patterns. This is supported by a standard interface that establishes the protocol of communication between user intentions and compiler transformations. Such flexibility increases the control a programmer has over efficient program execution while promoting software productivity via object-oriented reuse, type parameterization, and composi-

2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)
0-7695-2644-6/06 $20.00 © 2006 IEEE

tion.

3 Conceptual Framework

In this section we provide a conceptual framework for distributions in Chapel, which we use throughout the paper for the specification of their high-level semantics. The lower-level mechanisms provided in the language for controlling the layout of distributed data will later, in Section 4.1, lead to a refinement of this framework.

3.1 The Chapel Abstract Machine

The Chapel Abstract Machine contains a *memory component* and a *processing component*. Each execution of a Chapel program on the abstract machine is associated with a region of its memory component, called the *execution locale set*, and an unbounded, dynamically managed set of *threads* in the processing component.

The **execution locale set** is a non-empty finite set of identical **locales** determined at the time program execution begins. Locales are units of uniform memory access to which data and threads can be mapped. Accesses of a thread to data are called **local** if thread and data are mapped to the same locale, else **remote**. In terms of performance metrics, a local access is assumed to incur less overhead than a remote access.

Our distribution model handles the program-controlled mapping of data and threads to locales, and in addition allows the specification of the locale-internal layout of data.

3.2 Domains

Domains represent a central element of Chapel, linking index sets, distributions, arrays, and iterators. They generalize features present in other languages, such as HPF's *templates* [15] and ZPL's *regions* [4]. Specifically, a **domain** is a first-class entity, whose principal aspects are:

- An *index set*, which is a finite set of names for identifying components of arrays. The index set of a domain can be a Cartesian product of integer intervals as in Fortran 90, in which case the domain is referred to as *arithmetic*. However, index sets in Chapel can be much more general and may include virtually every set in which an equality relation is defined, such as instances of a class representing nodes in an unstructured grid. While arithmetic domains are considered to be usually invariant over significant portions of a program (with controlled redefinition of their index sets allowed via special statements), *indefinite domains* provide for a dynamic manipulation of a domain's index set by allowing the insertion or removal of individual indices at any time during the execution of an algorithm. The regular and semi-static character of arithmetic domains furthers the generation of highly efficient target code for accesses to arrays associated with such domains, whereas indefinite domains offer a higher degree of flexibility to the user.

- A *distribution*, which specifies a global mapping from the domain's index set to locales in the execution locale set, as well as the local arrangement of indices and data within locales.

- A set of associated *arrays*, which are mappings from the domain's index set to component variables of a given type. Arrays are allocated according to the domain's distribution. Due to the generality of index sets and types in Chapel the notion of an array is a more powerful concept than its counterparts in traditional languages.

- *Iterators*, which are functions defined over the index set of a domain, can be used to control the execution of sequential and parallel loops.

While every domain has a well-defined index set at any time of its existence, its other components are optional. For example, a domain used only for accessing locales in a program may have the index set as its sole component. However, this singular case is of little interest; the major focus of this paper is on the association of domains with arrays, the distribution of domains and their arrays, and the efficient access to array elements.

3.3 Index Mappings

This section introduces a class of functions defined over finite, non-empty sets. We use them primarily for modeling mappings involving index sets of domains.

Definition: Let X, Y denote finite, non-empty sets.

1. An **index mapping from X to Y** is a total function mapping X to the powerset, $\mathcal{P}(Y)$, of Y.

2. f is called **proper** iff $f(x) \neq \phi$ for all $x \in X$.

3. For a proper index mapping, f, from X to Y, the set

$$f(X) := \{y \in Y \mid \exists x \in X : y \in f(x)\}$$

is called the **image of** X under f.

4. Given a proper index mapping, f, from X to Y, the **inverse mapping**, f^{-1}, is the inverse of the relation in $X \times Y$ defined by f: for all $y \in Y$: $f^{-1}(y) := \{x \in X \mid y \in f(x)\}$.

5. A proper index mapping is called **replication-free** iff
   $\mid f(x) \mid = 1$ for all $x \in X$. Such a function will be interpreted as a total mapping, $f : X \to Y$. If it is surjective, then f^{-1} defines a *partition* of X in the mathematical sense, where all elements of X mapped to the same $y \in Y$ belong to the same equivalence class.

3.4 Data Distribution for Domains and Arrays

A data distribution is defined for a domain, whose index set is being distributed, and a subset of the execution locale set, which serves as the target of the distribution. The core components of a distribution are two functions referred to as the *global mapping* and the *layout*, both of which are defined over the domain's index set. The global mapping is an index mapping from the domain's index set to the target locales, while the layout maps indices to locations of the locale associated with them via the global mapping, thus allowing the specification of locale-internal data arrangements.

The global mapping is the primary component of a distribution and must be explicitly specified any time a distribution is defined. In contrast, the specification of the layout may be left to the system, which provides a default layout for any global mapping. Explicit specification of the layout is required if sophisticated data representation strategies are to be applied, which are beyond the reach of automatic methods, such as for distributed sparse data structures. In such a case, the layout specification allows virtually complete control over the locale-internal allocation policy. This capability can provide a significant performance gain in situations, where there is a strong correlation between data access patterns and internal data structures reflecting properties of an application that cannot be automatically recognized by the compiler.

The data distribution defined for a domain is applied to all arrays associated with the domain and controls their allocation in memory.

3.4.1 Global Mapping

The global mapping of a domain to a set of target locales results in subsets of domain indices being associated with each target locale. In the following definition we assume a domain D with index set I, which is distributed to a non-empty subset, LOC, of the execution locale set.

Definition:

1. The **global mapping**, δ, of a data distribution for D is a proper index mapping from I to LOC.

2. D is called the **source domain** of the distribution.

3. LOC is called the **target locale set**; its domain is called the **target domain**.

4. The inverse, δ^{-1}, of the global mapping is called the **ownership function**.

5. Given $loc \in LOC$, $\delta^{-1}(loc)$ is called the **distribution segment** of loc under the global map δ.

The distribution segment of a locale loc under the global map δ specifies the set of all indices in the source domain, I, which are mapped to loc via δ. The image, $\delta(I)$, of I under δ is called the **actual target locale set** for the distribution. This must be a non-empty set. $\delta(I)$ may be a proper subset of LOC; in that case there exist empty distribution segments.

In general, different distribution segments may contain identical elements. However, if the global mapping is replication-free, then the distribution segments associated with the actual target locale set are pairwise disjoint and constitute a partition of I.

Let A denote an array associated with domain D. Then the mapping $i \mapsto \delta(i)$ for all $i \in I$ determines the set of all locales on which the array component variable $A(i), i \in I$, is to be allocated. For every loc in the actual target locale set, $\delta(I)$, the **local array segment** of loc for A is the representation of the portion of A, $A(\delta^{-1}(loc))$, which is associated with the distribution segment of loc. The way in which data are stored in the local array segment is determined by the layout and the array's element type.

In many cases of interest, the global mapping of a distribution is replication-free. However, there are instances where the mapping to a powerset is required. An example is the replication of a "small" data structure (such as a scalar) to *all* locales: $\delta(i) = LOC$ for all $i \in I$. In this as well as in related cases, the decision to replicate is motivated by the goal to reduce communication latency and bandwidth requirements.

3.4.2 Layout

The **layout** of a data distribution specifies the locale-internal representation of distribution segments and array data in the context of a global mapping. Related functions include methods for accessing data and iterating over index sets. A more detailed discussion of the interface presented to the user for specifying layout will be found in Section 4.1. Here we introduce one basic function, which we call the *layout mapping*:

Definition: Let D and I be given as above, and δ denote a global mapping. The **layout mapping** of a data distribution is an index mapping, λ, from I to the set of indices [1] in the locale determined by the global mapping.

For any array A and index $i \in I$, $\delta(i)$ and $\lambda(i)$ are the key components determining the location in which element $A(i)$ is stored.

4 Use and Definition of Distributions in Chapel

```
const n1 = 1000000;
  ...
const D1C: domain(1) distributed(MyC())
                                    =[1..n1];

const D1B: domain(1) distributed(MyB())
    on Locales(1..num\_locales/10)=[1..n1];

var A1: [D1C] float;
var A2: [D1B] float;
...
```

Assuming a distribution to be given, it can be applied in a way similar to that in HPF-style languages, except that distributions in Chapel are bound to domains, and all arrays associated with a domain inherit the index set and distribution of the domain. The following code illustrates these relationships with a simple example:

```
const n1 = 1000000;
  ...
const D1C: domain(1) distributed(MyC())
                                    =[1..n1];

const D1B: domain(1) distributed(MyB())
    on Locales(1..num\_locales/10)=[1..n1];

var A1: [D1C] float;
var A2: [D1B] float;
...
```

We assume that `MyC` and `MyB` are distribution classes defined elsewhere. `D1C` and `D1B` are declared as invariant arithmetic domains of rank 1, with identical index sets defined as the set of all integers in the interval 1..1000000. `D1C` is distributed using an instance of the distribution class `MyC`. As we will see below, this represents a cyclic distribution. By default the target locale set of this distribution is the full execution locale set.

`D1B` is distributed using an instance of the distribution class `MyB`, which will later be defined as a block distribution. In contrast to `D1C`, only the first 10% of locales in the execution locale set (which is represented by the predefined variable `Locales`) are used as target locales in this case. Finally, `A1` and `A2` are arrays of floating point numbers whose index sets and distributions are respectively determined by the associated domains `D1C` and `D2C`.

Usually, standard distribution classes such as those represented by `MyC` and `MyB` will be defined as part of a distribution library. In the simplest case, such definitions could be specified as shown bellow:

```
class MyC: Distribution {
  const ntl: integer;
  function map(i:index(source)): locale
        {return Locales(mod(i-1,ntl)+1);}

  iterator DistSegIterator
      (loc:index(target)): index(source) {
    const N: integer = getSource().extent;
    const k: integer = locale\_index(loc);

    for i in k..N by ntl { yield(i); }
  }

  function GetDistributionSegment
            (loc: index(target)): Domain {
    const N: integer = getSource().extent;
    const k: integer = locale\_index(loc);

    return (k..N by ntl)
  }

}

class MyB: Distribution {
  const bl=...;
  /* global map for a simplified regular
     block distribution with block
     length bl: */
  function map(i: index(source)): locale
              {return Locales(ceil(i/bl));}

...
```

[1] In this context, we speak informally of *locations*.

}

Assume `num_locales=1000`. The distribution classes `MyC` and `MyB` are both introduced as subclasses of the base class `Distribution`. `MyC` can be used to distribute one-dimensional arithmetic domains cyclically, while `MyB` represents a regular one-dimensional block distribution. For `MyC`, the global mapping function, `map`, defines a replication-free mapping from each index $i \in 1..n1$ to the locale `Locales(mod(i-1,ntl)+1)`. For example, assuming that the number of locales is given as `ntl=1000`, index 876543 is mapped to locale `Locales`(543), whose distribution segment is given as

$$\delta^{-1}(543) := \{i \mid i = 543 + j * 1000, \text{ for all } j \text{ such that } 0 \leq j \leq 999\}$$

Subsequent calls of the iterator `DistSegIterator` yield the indices belonging to a locale, `loc`, while the function `GetDistributionSegment` determines the associated set of all indices.

Similarly, the distribution of `D1B` is specified by an instance of `MyB` and evaluated in the context of the target locale set `Locales`(1..100). The global mapping function in this case determines the block length, `bl`, as `bl=1000000/100`, yielding the value 10000, and resulting in the mapping

$$\delta(i) := \lceil \tfrac{i}{10000} \rceil \text{ for all } i, 1 \leq i \leq 1000000$$

The distribution segments are given as

$$\delta^{-1}(loc) := i \mid (loc-1) * bl + 1..loc * bl \text{ for all locales } \texttt{Locales}(loc);\ 1 \leq loc \leq 100.$$

4.1 User-Defined Specification of Distributions

The Chapel programmer does not necessarily need insight into the inner workings of a distribution. As long as the functionality provided by predefined distributions is sufficient, the programmer only needs knowledge of the interfaces to these distributions (which must be contained in a library) and guidance for their effective use in view of the array declarations in the program and the related access patterns in algorithms.

In this section we provide an overview of the interface and the methodology for the specification of user-defined distributions. In a sense, these features can be considered to be at a lower level of abstraction than the rest of the Chapel language since their implementation interacts directly with relevant properties of the hardware architectures on which Chapel programs will run. Specifically, the locality properties of a program execution and the required communication depend on the access patterns to arrays and the distributions of their domains. We can think of specialized *distribution writers* (which of course can be the application developers themselves) being in charge of developing sophisticated distribution libraries that reflect properties of applications.

We begin by explaining the interface of the distribution framework in Section 4.1.1. As already discussed, distributions can be essentially specified at two levels, one of which deals only with the global mapping while the other uses in addition an explicit layout to control in detail the arrangement of data at the locale-internal level. In the simplest case, the user is only required to specify the global mapping, while the system provides by default all functionality required for the allocation and management of distributions and distributed arrays. `MyB`, in the example introduced above, illustrates this option.

4.1.1 Main Classes

The distribution framework provides the distribution writer with tools for supplying application-specific functionality to the compiler and runtime system via a set of predefined public base classes with an overridable interface.

The base classes involved include `Domain`, `Distribution`, and `LocalSegment`, which are shown in the following code excerpt together with public methods:

```
class Domain {
 iterator for(): IndexType;
 iterator forall(): IndexType;
 function GetDistribution(): Distribution;
  /* the function GetParent yields the
     parent if the given domain is a
     subdomain, and nil else: */
 function GetParent(): Domain;
 function extent(): integer;
}

class Distribution {
 function getSource(): Domain;
 function getTargetDomain():Domain;
 function getTargetLocales():[target] locale;

 function map(i: index(source)): locale;
 iterator DistSegIterator(loc:index(target)):
                                  index(source));
 function GetDistributionSegment
              (loc: index(target)):Domain;
}

class LocalSegment: Domain {
 function getLocale(): locale;
 /* locale associated with an instance
```

```
                                    of this class */
  var LocalDomain: Domain;
        /* local data domain */
  function layout(i: index(source)):
                              index(LocalDomain);
}
```

We provide here only a reduced list focusing on the main functionality required in this section, and omitting details that will be used and explained in examples. For example, an arithmetic domain provides methods that determine the extent of its index set, and for each of its dimensions the lower and upper bounds.

The `Domain` class supports the built-in features for domains in the language. The core component of the framework is the `Distribution` base class. Any user-defined distribution class is a subclass of `Distribution`. The `getSource` method yields the source domain of the distribution, while the `getTargetDomain` and `getTargetLocales` methods respectively yield the target domain and the target locale set. The `map` method represents the global mapping from source indices to target locales. The iterator `DistSegIterator(loc)` produces the elements in the distribution segment associated with locale `loc`, as a sequence of source domain indices. Finally, the function `GetDistributionSegment`, applied to a locale `loc`, yields the domain associated with the distribution segment of `loc`.

The global mapping is the only method that must be always specified by the distribution writer when specifying a new distribution. The system provides default versions of `DistSegIterator` and `GetDistributionSegment` by automatically inverting the `map` function. These potentially expensive computations can be overridden by the user, as illustrated in one of our examples.

The `LocalSegment` class, a subclass of `Domain`, is a predefined class that provides a default representation of distributions and associated array data in locales. A separate instance of this class is created on every locale in the target locale set of a distribution. This class can be overridden if the user wants explicit control of either mechanism.

The function `getLocale` yields the locale for a specific instance of `LocalSegment`. Let `loc` denote such a locale: we discuss the properties of the particular instance of `LocalSegment` for this locale. The value of the public variable `LocalDomain` is the domain for the representation of local array data in locale `loc`. Each array associated with the source domain of the distribution is represented in locale `loc` by a separate array over the domain `LocalDomain`. As illustrated in the sparse matrix example in Section 4.1.2, the user may need to generate a set of persistent auxiliary data structures in each locale to support an efficient representation of the distribution and the mapping from global to locale-internal indices. Finally, the `layout` function maps a global source index (that in the given context can be assumed to be in the distribution segment associated with `loc`) to the associated index in `LocalDomain`.

4.1.2 Distributed Sparse Data Structures

In terms of building the distribution, the generation of a sparse structure differs from that of a dense domain in at least the following points:

- It is necessary to deal with two domains and their interrelationship: the algorithm writer formulates the program based on the original dense domain, i.e., indexing data collections in the same way as if they were dense. In contrast, the actual representation of the data and the implementation of the algorithm are based on the sparse subdomain of the dense domain.
- In many approaches used in practice, the distribution is determined in two steps:
  1. First, the dense domain is distributed, i.e., a mapping is defined for *all* indices of that domain, including the ones associated with zeroes. In general, this will result in an irregular partition, reflecting the sparsity pattern and communication considerations.
  2. Secondly, the resulting local segments are represented using a sparse format, such as CRS (compressed row storage).

The approach for user-defined distributions in Chapel presented so far is powerful enough to deal with this problem. For example, the required auxiliary data structures, such as the data, row, and column vectors in a compressed row storage (CRS) representation of distribution segments can be declared as persistent data structures that can be accessed by the layout function and iterators. Due to space limits details cannot be discussed in this paper.

5 Conclusion and Future Research

We have presented the design and implementation of language constructs and interfaces for user-defined distributions in Chapel, an explicitly parallel programming language. Our work is motivated by the need to provide better language and system support for productively writing parallel programs.

Our work exploits powerful Chapel concepts such as domains, generalized arrays, and iterators as well as the capability to distribute domains and their associated arrays across locales. The user-defined specification of a distribution includes a global mapping of indices to locales, and can, in addition, control the on-locale layout reflecting special allocation policies. Such allocation requirements can be expressed through a specialization of the `LocalSegment` class. The compiler automatically transforms locality-aware Chapel code into explicitly distributed code. We believe that the resulting programming model is systematic and concise, enabling reuse and high productivity. We also believe that the systematic exposure of distribution aspects to the programmer results in increased potential runtime efficiency and enables code optimizations.

A number of issues relevant in the context of distributions have not been discussed in this paper due to space and time limitations. These include the capability of user-defined distributions to specify general alignments, and the full specification of user-defined halos. There are also some areas in which research is currently in progress, including the efficient management of distributions under a set of domain operations defined in the language, the optimization of runtime support for the dynamic management of distributions and the dynamic optimization of communication for irregular problems.

The implementation of the language features described in this paper is currently underway. The presentation of the paper will include performance results of the Chapel distribution features, and comparisons with corresponding solutions based on the message-passing approach.

Acknowledgment

This paper is based upon work supported by the Defense Advanced Research Projects Agency under its Contract
No. NBCH3039003. The research described in this paper was partially carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

We would like to thank our collaborators and proponents of the Chapel language, David Callahan and Bradford Chamberlain of Cray Inc., for continuously providing ideas and constructive feedback, and exposing interesting issues related to distributions.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Jr. Guy L. Steele. Compiling fortran 8x array features for the connection machine computer system. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 42–56. ACM Press, 1988.

[2] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic Ideas for an object parallel language. *Scientific Programming*, 2(3), 1993.

[3] David Callahan, Bradford Chamberlain, and Hans Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60. April 2004.

[4] Bradford Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.

[5] Barbara M. Chapman, Piyush Mehrotra, and Hans P. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[6] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. ICC++ — A C++ dialect for high performance parallel computing. *Springer LNCS*, 1049:76–94, 1996.

[7] Ron Choy and Alan Edelman. Parallel MATLAB: Doing it Right. http://www.interactivesupercomputing.com/downloads/pmatlab.p

[8] Cray Inc. *Chapel Specification 4.0*, February 2005. http://chapel.cs.washington.edu/specification.pdf.

[9] Alain Darte, John Mellor-Crummey, Robert Fowler, and Daniel Chavarra-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *J. Parallel Distrib. Comput.*, 63(9):887–911, 2003.

[10] Yuri Dotsenko, Cristian Coarfa, and John Mellor-Crummey. A multi-platform co-array fortran compiler. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 29–40, Washington, DC, USA, 2004. IEEE Computer Society.

[11] E.Allen, D.Chase, V.Luchangco, J.-W. Maessen, S.Ryu, G.L.Steele Jr., and S.Tobon-Hochstadt. The fortress language specification version 0.707. Technical report, Sun Microsystems, Inc., July 2005.

[12] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *3rd International Workshop on Language Runtimes, ACM OOPSLA 2004*, Vancouver, BC, October 2004.

[13] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *J. Parallel Distrib. Comput.*, 50(1-2):61–82, 1998.

[14] G. Fox, Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W., and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Houston, TX, December 1990.

[15] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, January 1997.

[16] H.Meuer, E.Strohmaier, J.Dongarra, and H.D.Simon. Top 500 supercomputer sites. http://www.top500.org.

[17] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the berkeley upc compiler. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 63–73, New York, NY, USA, 2003. ACM Press.

[18] H.Zima, P.Brezany, B.Chapman, P.Mehrotra, and A.Schwald. Vienna fortran – a language specification. Technical Report 21, ICASE, NASA Langley Research Center, March 1992.

[19] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[20] Indiana University Pervasive Technology Labs. High performance java. http://www.hpjava.org.

[21] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1991.

[22] Robert E. Millstein. Control structures in illiac iv fortran. *Commun. ACM*, 16(10):621–627, 1973.

[23] P.Charles, C.Grothoff, V.Saraswat, C.Donawa, A.Kielstra, K.Ebcioglu, C.von Praun, and V.Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 519–538, 2005.

[24] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.